University of Groningen

Building Product Populations with Software Components

Ommering, Robbert Christiaan van

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2004

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Ommering, R. C. V. (2004). *Building Product Populations with Software Components*. s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 30-05-2023

Chapter 2

Formalizing Software Architecture

Published as: *Languages for Formalizing, Visualizing and Verifying Software Architecture*, Rob van Ommering, René Krikhaar, Loe Feijs, Computer Languages 27 (2001), p3-18.

Abstract: In this paper we describe languages for formalizing, visualizing and verifying software architectures. This helps us in solving two related problems: (1) the reconstruction of architectures of existing systems, and (2) the definition and verification of architectures of new systems. We define an expression language for formulating architectural rules, a graph language for visualizing various structures of design, and a dialogue language for interactively exercising the former two languages. We have applied these languages in a number of industrial cases.

2.1 Introduction

A well-defined software architecture is an essential element for complex software systems. Ideally, such an architecture is defined in advance, i.e. before the implementation is started (forward architecting). In practice, however, this is not always the case, and often an architecture needs to be reconstructed from an implementation (reverse architecting). Reverse architecting and forward architecting have certain problems in common. Consider the following two scenarios:

- *Reverse architecting*: a software architect is updating an existing software system to fulfill some new requirements. He wants to study the architecture of the software, but no explicit description is available. How can he *extract* the architecture from the implementation, and how can he *verify* that the induced architecture is indeed the one that the implementation satisfies?
- *Forward architecting*: a software architect has just created a *new* software architecture, by defining architectural concepts and rules, together with a decomposition of the software into layers, subsystems and/or components. A large team of software engineers is currently implementing the architecture.

© 2001 Elsevier Science Inc. Reprinted, with permission, from Computer Languages, 27, 2001, Rob van Ommering, René Krikhaar, Loe Feijs, 'Languages for Formalizing, Visualizing and Verifying Software Architecture', p3-18.

How can he ensure that they indeed follow the rules and decomposition? Can these rules be *formalized* and then be *verified* automatically?

We are concerned with the formalization of software architectures to automatically verify implementations against the architecture. This is useful in reverse architecting, to check whether an extracted architecture indeed matches the implementation. This is also useful in forward architecting, as experience shows that software engineers do not always obey the prescribed architecture. Verifying the implementation against the architecture can then improve the quality of both the architecture and the implementation, as in case of differences, either the architecture or the implementation can be assumed to be wrong.

The approach that we have developed employs three languages: an *expression language* that we use to formulate architectural rules, a *graph language* that we use to visualize design structure, and a *dialogue language* that allows us to experiment interactively with the other two languages. The expression language is based upon an algebra of binary relations in which partitions play an important role. The graph language provides visual representations of graphs with a variety of graphical attributes. The dialogue language uses a desktop metaphor with among other things a relational calculator and various 'drag and drop' techniques to allow for easy experimentation with the first two languages.

This paper is organized as follows. In the section 2.2, we provide a birds-eye view of our approach, based upon a simplified definition of software architecture. Sections 2.3-2.5 discuss the three languages mentioned above. We end this paper with some concluding remarks.

2.2 Formalizing and Verifying Software Architecture

Software architecture can be defined as *the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution* [39]. Basically, this implies that an architecture is an *abstraction of systems* [5]. In this paper, we define architecture as a *specification of a design*, where a design is an abstraction (read: the 'structure') of an implementation. Note that multiple designs can satisfy a single architecture; certain design details can be left as 'implementation freedom' for the designers. Note also that multiple implementations can share the same design; certain coding details can be seen as implementation freedom for the software engineers.

Our definition has operational semantics. An architecture can be defined in a 'legal document', and for every implementation, an independent party can establish whether it satisfies the architecture or not. In agreement with our definition, this proceeds in two steps (see Figure 6). First, the design structure *d* is abstracted from the implementation *i* using an abstraction function *f*. Second, this design *d* is verified against the architecture *a* using a 'satisfies' operator $\subseteq$.

Two domains play a role here, the implementation domain *I*, and the architecture and design domain *D*. We shall treat the architecture and design domain formally, but we shall deal with the implementation domain *I* in an ad-hoc way. Abstracting a design structure *d* from an implementation *i* is a mechanical process, which we implement in the form of a large number of extraction tools. Each tool processes the implementation and reconstructs one design aspect in the domain *D*. Each tool has an implementation that is specific for the design aspect and also for the implementation properties such as programming language and file system conventions. Note that although we do not have a full and formal model of the implementation, we do construct an accurate abstraction of the implementation, which we call the *design*.

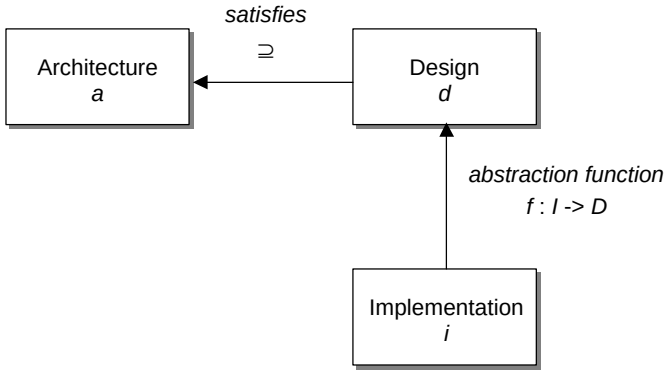

Figure 6. Relation between Architecture, Design and Implementation

Checking a design against an architecture can also be a mechanical process, at least for that part of the architecture that can be formalized. Note that architecture and design are expressed in the same language. In the next section we shall define this language and show an example of the operator $\subseteq$.

It is important to note that we have applied these techniques to those aspects of software architecture that are sometimes called the *modular architecture* [101], i.e. the static decomposition of a system in terms of layers, units, components, files, functions, et cetera. The techniques can in principle also be applied to more dynamical architectural structures, but discussion of that is outside the scope of this paper.

2.3 The Expression Language

The expression language was designed with the following requirements in mind:

- It should allow us to formalize design structures and architectural rules.
- It should be generic so that we can use it for multiple aspects of design and also in multiple application domains.

- It should be possible to build generic tool support for it, so that we can concentrate on software architecture and not on building aspect- or domain-specific tools.

Our expression language is an algebra based on sets, binary relations and a special kind of binary relation called part-of relations. This algebra is called the relation partition algebra (RPA). The difference between part-of relations and partitions is subtle and discussed in more detail elsewhere [29]. We shall give a brief overview of RPA in the next sections.

2.3.1 Sets

The algebra of *sets* is well known. We can calculate the union $S_1 \cup S_2$, the intersection $S_1 \cap S_2$, the difference $S_1 \setminus S_2$, and perform many other operations on sets. There are a large number of algebraic laws that describe the behavior of these operators. Some operations impose some problems, like the complement of a set, but since we always operate in finite and known universes, these problems are not fundamental.

We use sets to model the entities in modular architecture. Examples of such entities are functions, files, components, subsystems, layers, et cetera.

Creating tool support for set operations is straightforward: many libraries offer such facilities. In principle, sets can be created for different types of objects. Our implementation is simpler: we can only deal with sets of *atoms*, where an atom is an object with a unique name but without further content. If we want to assign other properties to objects, we use binary relations, as will be explained in the next section.

2.3.2 Relations

A binary *relation* R on sets S_1 and S_2 can be defined as a set of pairs of elements, i.e. $R \subseteq S * S$. As such, binary relations inherit all the algebraic properties from sets. We can for instance calculate the union $R_1 \cup R_2$, the intersection $R_1 \cap R_2$, the difference $R_1 \setminus R_2$, and perform many other operations. But there is a second side to relations. By viewing them as a mapping from one set to another set, we can calculate the inverse mapping R^{-1}, we can compose two relations into a new relation $R_3 = R_2 \circ R_1$ (R_2 applied *after* R_1), and we can calculate the transitive closure R^* of a relation R. Also, we can calculate the domain *dom*(R) and the range *ran*(R) of a relation.

The algebra of relations has also been studied intensively. A well-known application area is relational databases. The corresponding query languages form in fact an implementation of the operations on relations. A large part of our work on architecture formalization and verification can indeed be implemented in the form of a database (containing the intended and extracted design information) and

queries on the database (formalizing the architectural rules). We shall come back to this in section 2.6.2.

We use binary relations to model various structures in software architecture. Examples are the call relation (or call graph) between functions, the include relation between files, and the use relation between modules or components. Also we model the decomposition of a system with various part-of relations, such as 'functions being part-of files', 'files being part-of components', et cetera. Thirdly, as mentioned in section 2.3.1, we use relations to add properties to objects. A file we usually represent as an atom with as name the path of the file. Other properties, such as a time stamp, are represented as binary relations (actually a function) that relate the files to time stamps.

Our own implementation of relations is based on pairs of *atoms*, the same atoms as mentioned in section 2.3.1. This allows us to handle very large relations, say in the order of 10^6 pairs. Call graphs in industrial systems often have such sizes.

2.3.3 Part-of Relations

A special kind of binary relation is a *part-of relation* that defines which entities are part-of which other entities. In this paper, we only consider functional and acyclic part-of relations. A typical part-of hierarchy in software architecture contains functions, files, components, subsystems, layers, and the system as root. Part-of relations inherit all operations from sets and relations, so again we can take the union $P_1 \cup P_2$ of two part-of relations, and also compose part-of relations: $P_2 \circ P_1$. But the specific interpretation of a relation as a part-of relation also gives rise to new operations such as lifting.

We define the *lifting* operator $\uparrow$ as follows: $R \uparrow P = P \circ R \circ P^{-1}$. The interpretation behind this is the following. If $R \subseteq F*F$ is a call graph on the set of functions F, and $P \subseteq F*M$ is a part-of relation that defines in which module M a function F is defined, then $R \uparrow P$ is the induced usage relation between modules. In other words, $R \uparrow P$ represents the following *uses* relation:

> A module m_1 *uses* a module m_2 if and only if there is a function f_1 in m_1 and a function f_2 in m_2 where f_1 *calls* f_2.

An example of lifting can be seen in Figure 7. The left-hand side of this figure shows the functions *main, a, b, c* and *d*, and the modules *Appl*, *DB* and *Lib*. Through nesting is shown which functions are part-of which modules. Arrows between functions denote the function call graph. The right hand side shows only the modules, and the usage relation between the modules, obtained by lifting the call graph using the part-of relation.

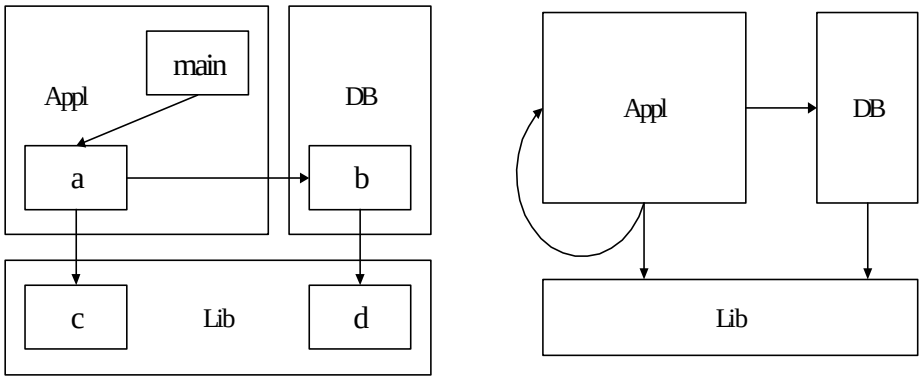

Figure 7. Lifting functions calls to module level

2.3.4 Architectural Rules

The example in Figure 7 allows us to introduce an example of an architectural rule. Suppose that the right-hand side of Figure 7 represents the allowed usage relation U between modules, as defined by software architects. The left-hand side represents the actual call relation C between functions, and the part-of relation P. The architectural rule states that any call between functions should obey the allowed module usage relation. The rule can be formalized as follows:

$$C \uparrow P \subseteq U$$

Note that module *Appl* uses itself in the right-hand side of Figure 7. In general, we are not interested in function calls *within* a module, so the architectural rule can be reformulated as:

$$(C \uparrow P) \setminus I \subseteq U'$$

where I is the identical relation on modules, and U' is the allowed usage relation but without self references.

2.4 The Graph Language

Design structures can be represented as *directed graphs*. In this section we define graphs and views on graphs. We also show how we can extract design structures from implementations and visualize them.

2.4.1 Graphs and Views on Graphs

We define a *graph* G as a tuple (S, R), where S is a set of objects and R is a relation on S*S. Similar to the explanation in section 2.3.1, objects are identified by their name, which is a simple string of characters. Apart from their name, objects have no further content.

We define a *view* V on a graph G as a tuple (S', R', L) where:

- $S' \subseteq S$
- $R' = R \lceil_{car} S'$
- L is a layout function that assigns a position to each element of S.

The symbol $\lceil_{car}$ means *carrier restrict*, and produces that subset of the relation R that contains all pairs in which the left- and right-hand side are both element of the specified S'. In a formula:

$$R \lceil_{car} S = R \cap (S*S)$$

A view is a *visualization* of *part of* a graph. The visualization is obtained as follows:

- For each object in S' a rectangle with standard size and shape is drawn at the position specified by the layout function. The object's name is drawn inside the rectangle.
- For each pair of objects in the relation R', a straight line is drawn that connects the centers of the rectangles (but the line is drawn *behind* the rectangles).

An example view of part of a design structure is shown in Figure 8.

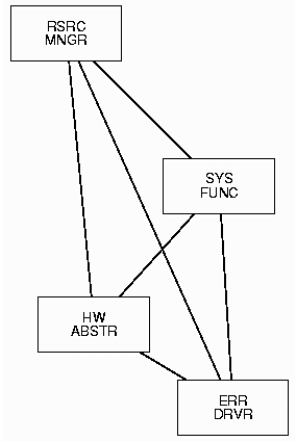

Figure 8. A simple graph showing a design structure

It is interesting to note that we demand a certain completeness of our views. Although the set of objects in a view may be any subset of objects in the corresponding graph, we show a line between objects if and only if the pair of objects is in the relation. Therefore, we never hide a line between objects just to 'clean-up' the picture, a technique often applied when drawing informal pictures of designs. But we can systematically filter out certain connections, using the operations of RPA. An example showing this is beyond the scope of this paper.

A second remark is that in some cases we draw lines instead of arrows between the boxes. Many relations in software architecture are (or should be) a-cyclic. When we draw such relations, our 'normal' drawing direction is top-down. If an edge is to be drawn in upward direction, it is marked in a special way, e.g. by making it thicker. This allows us to spot anomalies in our relations easily. Note that the architectural diagrams drawn in [105] are similar to our pictures.

2.4.2 Teddy

We have built a tool called Teddy [68] that takes a view V consisting of a set D, a relation R and a layout L as input, and that visualizes this view. Figure 9 shows an example use of this tool. As a side note, Teddy uses the term *domain* for sets in its object model for implementation reasons (*Set* is a keyword in the language in which we have implemented Teddy).

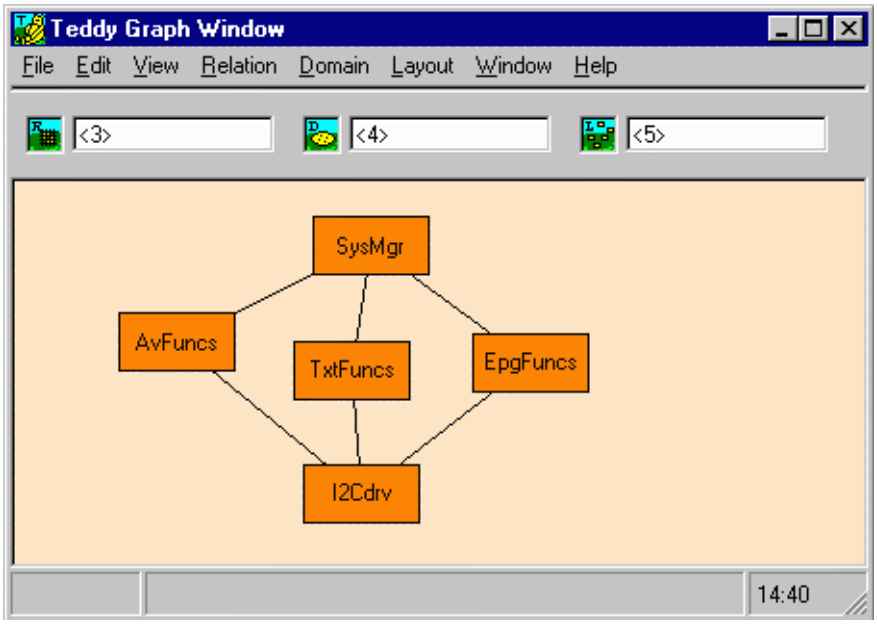

Figure 9. Example use of Teddy

In Figure 9, the three input fields marked R, D and L contain the actual relation, the set and the layout being viewed. Teddy allows the relation R and the layout L to be 'larger' than the set S, as it performs internally a carrier restrict operation (as explained in section 2.4.1) on R and a domain restrict operation ($L \cap (S^*ran(L))$) on L before visualization. So in Figure 9, both the relation and the layout function may contain more objects than the five objects shown. Widening the precondition of Teddy is a practical measure to enhance its usability.

Teddy can also be used to edit the set D, the relation R and the layout L. Dragging a rectangle will change the layout L. Through menus, objects can be added to or

removed from the view, thus changing the set D. Finally, lines between objects can be inserted or deleted, thus changing the relation R. Changing S and L is a frequent activity during architectural verification. The relation R, on the other hand, is usually derived from an implementation and need not be changed (but this does not hold for all relations).

2.4.3 Decorating the Graph

There are various ways in which we can enhance the visual representation.

First of all, we use the *shape* of the objects in the visual graph to represent the kind of object in the design. For instance, we make a distinction between value-based components (containing simple data types, stateless services such as string compare, et cetera) and state-based components. This allows us to visually concentrate on the latter, and ignore the former in certain design discussions. Sharing stateless components is usually harmless, while sharing components (or objects) with state can result in many forms of inconsistency. We used rounded rectangles for value-based components, and normal rectangles for state based components.

We used *dashed lines* for edges to objects representing value-based components, and *solid lines* for edges to state based components, again to diminish the importance of the former components in the design discussions.

We used *thick borders* for components that are used by components outside of the view, and *thin borders* for the rest. This allowed us to create a view of a *subsystem* in the architecture, and to study the export signature of the subsystem. We could have done the reverse as well, and define a special notation for components that *use* external components versus components that don't, but we did not find it very useful (yet).

We defined the *height and width* of individual blocks (this is an extension of the layout function), to make the pictures correspond more closely to the ''original' design pictures.

Finally, we generalized the usage relation to a *multi relation* (with a usage count per pair of objects - this will be explained further below), and visualized the count using the *thickness* of the lines. This allows us to distinguish between 'important' and 'less important' usage between components (though it should be said in all fairness that sometimes a single call to an important function has more impact on the architecture than a large number of calls to less important functions).

The extensions grew directly out of our efforts in practical software development projects to address immediate needs of the software architects. More extensions are feasible:

- different border types (solid, dashed, dotted),

- fill color and/or texture of shapes,
- border color,
- size of the shape (e.g. representing the code size).

2.4.4 Parameterizing Teddy

The extensions to the graphical language mentioned in the previous section could be hard-coded into Teddy, but of course it is much better if the tool is parameterized over these extensions. We therefore make a distinction between *logical* attributes (such as value-based versus state-based) and *graphical* attributes (such as shape), and allow the user to define the mapping.

One technique for doing this is to use labeled graphs, where each node and edge can have multiple labels. We have chosen for a slightly different approach, which is in line with the expression language discussed in Section 2.3. Graphical attributes of objects are controlled by extra relations that map an object to for instance a shape. Graphical attributes of edges are controlled by using multiple relations, one for each edge type. Although slightly more cumbersome for users, limiting ourselves to sets and binary relations reduces the complexity of the expression and dialogue language.

2.5 The Dialogue Language

Languages are systematic ways of expressing interaction, in our case between users and computers. We have defined the expression language (RPA) for representing design structures and architectural rules, and the graph language for visualizing structural information. The third language, the dialogue language allows to use RPA interactively to create design views, and also to replay the calculations when new data becomes available. An RPA calculator is the dominating tool in the dialogue language, but also other tools such as various viewers are part of the language.

2.5.1 The RPA Calculator

The RPA calculator is a desktop interactive calculator that can execute operations on sets and relations. We used the Reverse Polish Notation paradigm, so the calculator has a small stack of sets and relations, and each operation takes zero or more elements from the stack and pushes the result back on the stack. The stack is made visible on the screen (see Figure 10).

Many operations require specific types of arguments and produce specific types of results. For example, the *domain* operation requires a relation as argument and it produces a set as result. And the *union* operation either requires two sets and produces a set, or it requires two relations and it produces a relation. Therefore,

depending on the types of arguments on the stack, buttons in the calculator are enabled or disabled (both functionally and visibly).

Sets and relations can be stored in files and can be loaded from files. We use a simple file format. A set is stored as a file where each line contains the name of an element. A relation is stored as a file where each line contains the names of two elements in a pair, separated by a space. Choosing a simple format makes the writing of scripts to generate and process the files easy.

Drag and drop is also supported in the calculator. A file can be dragged from e.g. an explorer window and dropped at any (visible) location of the stack. Also, any (visible) element of the stack can be dragged and dropped in an explorer window. The drag and drop paradigm can be used to exchange data between different instances of the calculator, and also between the calculator and the other tools, as discussed in the next sections.

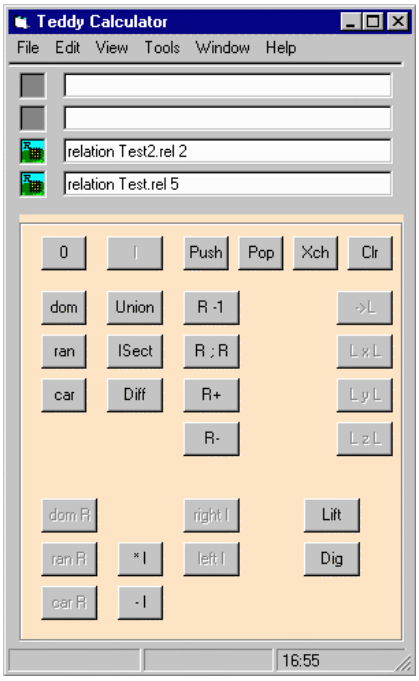

Figure 10. The RPA interactive calculator

2.5.2 The Graph Viewer

We already discussed the graph viewer Teddy in section 2.4.2. The relation, set and layout used by the tools are shown as icons in a tool bar. Using drag and drop, results can be transported from the calculator to the tool bar (and vice versa if necessary). This allows rapid experimentation with data extracted from applications, until the right visual representation is obtained. Needless to say, the

graph viewer can be instantiated as many times as desired. For instance, it is a matter of a few clicks and drags to create a second window that displays the transitive closure of a relation viewed in the first window (shown in Figure 11).

2.5.3 Calculating Layouts

The calculator can perform operations on sets and relations, but Teddy also needs layouts. Therefore, we have implemented a small set of operations on layouts as well. Note that a layout is essentially a mapping of elements to two-dimensional position. It therefore inherits operations on relations and sets, such as *domain* and *domain restrict*.

The operation *union* on layouts needs a further explanation. One aspect is that a layout relation is functional, and so must the union of two layouts be (otherwise an object would have two positions). Another aspect is the location of the resulting objects: merely merging views will probably cause objects to overlap. We therefore defined different flavors of union by considering the bounding boxes of two layouts and stacking them horizontally or vertically, and top/center/bottom (respectively left/center/right) aligned.

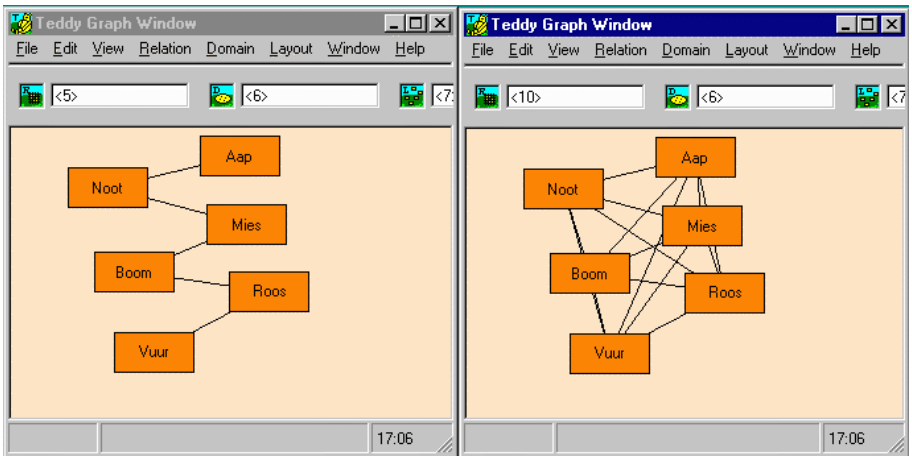

Figure 11. Two graph windows, one showing a relation, the other the transitive closure of this relation

The different union operations allow us to create views of parts of a usage relation by manually adjusting the layout using Teddy, and then to use the union operation to calculate layouts for larger parts of the relation. We can thus create a full overview of a large system step by step.

2.5.4 The VRML Viewer

Drawing graphs in two dimensions will soon result in lines crossing each other. One way of dealing with this problem is by extending the view in the third dimension. We have a special 3D viewer that enables this.

Instead of building 3D viewers ourselves, we rely on existing technology by generating VRML [108] code and viewing that in an appropriate viewer. Of course, we need a 3D layout to generate VRML. We obtain these by generalizing our union operations on layouts into the third dimension, e.g. by also providing a stacking operation in the z-direction. An example can be seen in Figure 12. For more details on our work we refer to [28]. We are not the only ones to represent architecture in with three-dimensional diagrams. See [36] for another 3D visualization of a design structure.

2.5.5 The Module Interconnection Browser

A usage relation can be shown as a graph, but also as a matrix with the elements of domain and range labeling the horizontal and vertical axes, and with ticks in the cells to indicate the pairs in the relation. Moreover, the horizontal and vertical axes can be tree views of the elements in a part-of structure, and the user can interactively expand and collapse nodes. Multi relations can be shown by putting the cardinality of a pair in a cell, instead of a single tick mark. See [14] for more information.

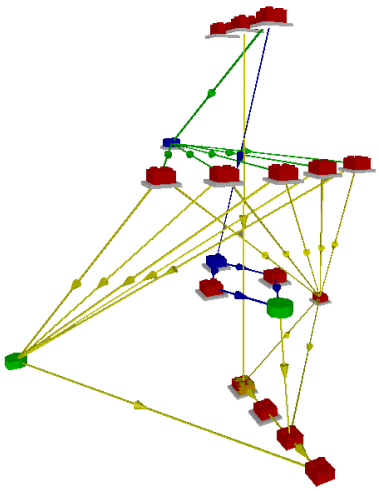

Figure 12. A three dimensional design structure viewer

2.5.6 Recalculating Expressions

The simple use of the calculator and the various viewers involves extracting structural information from an implementation, manipulating this information

using RPA, and viewing the results using various viewers. Sometimes, we want to repeat the calculations with different inputs. To achieve that, the calculator remembers the operation together with the result, so that it can re-evaluate the operation if the inputs change. Also, when interactively dragging results from the calculator to the viewers, it is possible to specify whether to drag *by value* (i.e. an (unlinked) copy of the result is created) or *by reference* (the result is liable to recalculation). We shall give some examples.

Suppose we want to view different relations with the same layout. We can drag the layout used in one instance of the graph viewer *by reference* to the layout icon of another instance of the graph viewer. If we now update the layout in any of the windows interactively, then the other window will be updated automatically.

Suppose that we are viewing a relation in one window, and we use the calculator to calculate the transitive closure of the relation and show it in another window (see Figure 11) using the same layout (and the same subset of nodes). The layout can now be changed in the first window, and the layout of the second window changes immediately. If an edge is added to the first window, the transitive closure is recalculated and immediately displayed in the second window.

As another example, we show a relation in two graph view windows and one 3D window. The two graph view windows show disjoint subsets of the elements, the 3D window shows all elements. The 3D layout is calculated by stacking the 2D layouts in the third dimension. We can now use the two graph view windows to quickly edit the 3D layout.

2.6 Concluding Remarks

2.6.1 Applicability

In this section we want, first of all, to look back and see how the described languages satisfy the needs of software architects as sketched in the introduction. We have the expression language, the graph language, and the dialogue language. We may consider the dialogue language as an extended and interactive combination of the expression language and the graph language.

Both in forward and in reverse architecting it is important to *formalize* rules and *verify* them automatically. A typical formalized rule is of the form R `is-a-subset-of' E, which is the same as R \ E = ∅ (using the RPA operator `\' for relational difference and using the constant ∅ for empty relation). Here we have an expression for the real relation R, as obtained by extraction and further calculations from the source code under development, next to an expression for the expected relation E (for example E may tell which layers are allowed to make direct usage of certain other layers). The verification is done by first extracting R. Most details of this extraction are outside the scope of the present paper and we just refer to [27] for more details. The next step in the verification is to evaluate R and E, perform

the calculation of `\' and look whether the result is empty. Although, in principle, this concludes the verification, we learned from practice that usually the result is not empty and then the user wants to find out more: which pairs in the relation are *violations* of the rule, and *why?* To see the *violations* and to trace them back to the overall design is precisely where the graph language comes in, and to formulate the usual sequence of additional questions needed to find out *why*, is where the dialogue language comes in.

The approach has been used in several Philips projects, both in the research department and in the industrial divisions. Amongst the analyzed systems we mention television systems, several telecommunication systems and also medical systems. A number of these cases are described in [27] en [46].

2.6.2 RPA versus Databases

We already mentioned the close relation between the use of relational databases and RPA in the field of architecture formalization and verification. There are two points where RPA provides advantages:

- *theoretical*: by considering the special role that part-of relations (usually a tree and not a graph) play in our analysis (e.g. lifting), we obtain a specialized set of operations and properties called Relation Partition Algebra.
- *implementation*: we often have sets and relations with an extremely high cardinality, and it pays to create a dedicated implementation of the operations, instead of relying on standard data base technology. Also, databases are usually not strong in calculating the transitive closure, which is something we regularly use.

Apart from these, the use of RPA is very extensible: one can just add relations and rules without having to change the model. For more information on the pure mathematics of Relation Partition Algebra, we refer to [29]. In this paper, we only discuss the language-oriented aspects and their applications.

2.6.3 Related Work

For reverse engineering, we can refer to earlier publications of ourselves that focus on the software architecture aspects [27], [46] and on the algebraic laws of our expression language [29]. Similar work on reverse engineering includes the following:

- Kazman and Carriere [44] developed a workbench to support the extraction and fusion of architectural views. Both *uses relations* ('functions call functions') and *part-of relations* ('classes define functions', 'files contain functions') are recognized. The architectural views are similar to the graph language of Section 2.4.

- Chikofsky and Cross [20] give an overview of the field of reverse engineering and design recovery. Their terminology is most helpful, for example to distinguish between *reverse engineering* (analysis and abstraction) and *re-engineering* (renovation and alteration).
- Holt [35] uses operators based on Tarski's relational algebra, supported by a language called Grok. Grok scripts are similar to the expression language of Section 2.3.
- Murphy at al. [63] use high-level abstractions called *software reflexion models*, which can be used to determine where the engineer's high-level model does and does not agree with the source model. Operations such as lifting are used in an implicit way. The reflexion model itself is formalized in Z.

For forward architecting, one could argue that it is better to define the software architecture in an *architecture description language* (ADL). Such a description can then be used to control the implementation build process so that no one can violate the architecture. Architectural description languages indeed exist (Aesop [1], Darwin [52], Koala [72], Rapide [102]), but they are not commonly used yet, and if used, they concentrate on decomposition only. General architectural rules may be built into the semantics of the ADL, but domain specific rules usually cannot be added. Furthermore, you might want to tolerate local and/or temporary deviations of the architecture, but a rigid ADL compiler might not allow that.

2.6.4 Acknowledgements

The authors want to thank Piërre van de Laar and André Postma for their contributions to this paper.